Chapter 1

Model Clone Detection and its role in Emergent Model Pattern Mining

Towards using Model Clone Detectors as
Emergent Pattern Miners - Potential and Challenges

Matthew Stephan* and Eric J. Rapos*
**Miami University, Department of Computer Science and Software Engineering*
510 E. High St., Oxford, OH 45056

ABSTRACT
Model-based software engineering approaches continue to gain traction in both industry and research. Accordingly the size, complexity, and prevalence of the models themselves are increasing. Model analysis and management thus becomes an essential task within any model-based process. One form of analysis that can support model-based approaches during the software engineering life cycle is pattern extraction, whereby tooling identifies emergent model patterns. These patterns can be used by analysts to ensure adherence to standards, software quality assurance, and library generation and optimization. In this chapter, we discuss the plausibility of using model clone detection as a form of emergent pattern mining for model-based systems. After a brief primer on the field of model clone detection and model pattern detection, we propose a conceptual framework, MCPM, centered on model clone detection that analysts can employ to detect emergent patterns in their models. In describing our framework concept, we illustrate our ideas using Simulink, and our Simulink model clone detector, Simone, as an example. However, we also consider other model clone detectors' potential within the MCPM framework. This includes how existing research and tooling can be applied to each step within the framework. We identify open challenges for researchers in realizing model clone detection as a model pattern mining tool, as well the potential benefit that can be experienced by practitioners in the application of MCPM.

KEYWORDS
model clone detection, model clones, model pattern mining, model patterns, model driven engineering

1.1 INTRODUCTION

As the proliferation of model driven engineering continues [77], there is an emergence and continuous flow of software models. This includes model repositories, for example MDEForge [10] and the Lindholmen Dataset [32]; common open source repositories that contain models, such as GitHub and SourceForge;

and organizations' internal repositories. Source-code based techniques are not applicable nor suitable for model-specific analytics. While this, and the increasing scale and intricacy of software models, presents interesting challenges in repository management, model visualization, and other areas we address in this book, it gives rise to more advanced and learning-based opportunities for analytics.

One such opportunity is software pattern extraction based on a corpus of software models. Mining traditional software source code in general is a mature and growing field [30]. However, the mining of model driven engineering artifacts is only recently gaining traction as model-based approaches become more widespread, and the complexity and size of the artifacts has grown. For this chapter, we are interested solely in *emergent patterns*, which are recurring software solutions that occur organically within software projects that are not known to analysts beforehand. Such a facility can help in areas such as standards enforcement, quality assurance, software development and refactoring, maintenance, and others. Sharma et al., from NEC Labratories America, identify pattern matching as a key challenge in the field of cyber-physical systems (CPS) [59], which are often developed exclusively or predominately as software models [20].

One tool for model-driven engineering analysis is model clone detection [19]. Model clone detection is a form of model comparison [67] that involves analyzing models to identify identical and/or similar models with respect to some measure of similarity. Similarity analysis can include both structural and semantic aspects. Clone detection in software has many uses including estimation of maintenance costs, fault prediction, refactoring, and others [35]. In this chapter, we describe using model clone detection as an emergent model pattern miner within a conceptual Model Clone detection Pattern Mining (MCPM) framework. We describe model clone detection's role in the MCPM, including its potential and challenges; how existing work fits into the requirements of the MCPM framework; and provide examples. Our goal is to improve model analytics by helping pave the way for model clone detection to be used as an analytical tool that discovers emergent patterns.

We begin in Section 1.2 by providing the background material necessary to understand this chapter. In Section 1.3, we define our MCPM concept with a detailed description of the framework, its sub steps, examples, and a review of existing tools and research for each of its phases. We identify open challenges and future work related to the MCPM framework in Section 1.4, and conclude in Section 1.5.

1.2 BACKGROUND MATERIAL

In this section, we describe background material on software patterns in general with a focus on pattern mining for source code and models, and model clone detection. For the latter, we describe the different types of model clones, as that is an important consideration when describing our framework's requirements for MCPM.

1.2.1 Software Patterns

Software patterns are an integral part of software engineering. They can be thought of as successful, commonly employed, and validated approaches to solving a software engineering problem [58]. They are an abstraction that occurs in "non-arbitrary" and recurring contexts [52]. One of the most popular examples, both in education and practice, are design patterns [15], which describe established software development solutions to frequent design problems. The opposite of design patterns are anti patterns, which are commonly occurring incorrect/problematic ways that people solve problems using software [13]. The term 'patterns' can be used as an umbrella term for both, and we do so in this chapter unless we otherwise use the explicit "design" or "anti" descriptors. Software patterns exist also for many other domains and contexts. For example, software security patterns [24], agent-oriented domains [41], architectural patterns [29, 61], agile development [44], and others. Software patterns have many uses including software evolution [81], architecture evaluation [82], fault detection [21], and establishing traceability links [36]. Traditionally, software patterns have been formulated manually by industrial and academic experts based on real-life development experiences and struggles, a posteriori [75]. However, it also is possible to have unknown patterns detected through analysis of existing systems through the process of pattern mining. For this chapter, we are interested specifically in *emergent patterns*, which occur naturally within software projects and are not known to analysts before system evaluation.

1.2.1.1 Source Code Pattern Mining and Detection

Analysis of software systems to discover patterns allows analysts to identify and codify commonly occurring solutions and implementations within software systems. This can involve mining patterns from software revision histories [42], source code [8, 80], execution traces [55, 73], architectures [45, 82], and more. A common application of source code pattern mining is to detect the presence of known design patters. Dong et al. [22] identified seven different categories for design pattern mining through a survey of the literature: structural aspect mining; behavioral aspect mining; structural and behavioral aspect mining; structural and semantic mining; structural, behavioral, and semantic aspect mining; and pattern composition. Structural aspects refer to design relationships, such as aggregation and generalization. Balany and Ferenc [8] similarly mine design patterns, but do so by building an abstract semantic graph from C++ code. Shi and Olsson [60] detect design patterns in Java code by focusing on structure-driven and behavior-driven patterns from the popular Gang of Four design pattern list [15]. Tsantalis et al. reverse engineer source code into matrices in order to detect design patterns by looking at similarity scores between the design patterns and source code [74].

This work on design pattern detection is different than what we focus on in this chapter, as we are concerned with emergent patterns that are not known before mining takes place, whereas design pattern mining looks explicitly for those known patterns. Additionally, all these approaches require and work explicitly on source code, which is not always present in model-based development nor model-driven approaches.

1.2.1.2 Model Pattern Mining

Model pattern mining involves analyzing software models explicitly, instead of source code. This is beneficial and relevant for software organizations and domains that rely heavily, or exclusively, on model-based or model-driven ideologies. This often includes automotive, aerospace, telecommunications, and other formal and safety-critical domains. This can include modeling languages such as UML, SysML, Simulink, and others. Model pattern mining is considered challenging as it often involves the general subgraph isomorphism problem, which is NP complete [26]. Thus, model pattern mining techniques must account for this through heuristics, considering alternative forms of analysis, and other means. Paakki et al. analyze UML models using constraint satisfaction approaches to detect known architectural patterns, both good and bad [45]. Gupta et al. employ the state space representation of graph matching to find instances of existing UML patterns, represented as graphs, in other UML models [27]. Pande et al. [46] look for design patterns in UML diagrams of graphical information systems using graph distances. Bergenti and Poggi [11] detect the Gang of Four Patterns in UML models by using rule-based techniques, as do Ballis et al [9]. Liutel et al. [43] use answer set programming, a declarative language, to detect patterns and anti patterns using both facts and rules on those facts. Fourati et al. use metrics to detect anti patterns on the UML level by looking at the amount of coupling between objects, the number of methods that can be invoked by a class, cohesion attributes, and complexity attributes such as imported interfaces [25]. Wenzel performs a fuzzy evaluation of UML models to determine if a model is exhibiting structural properties indicative of a failed or incomplete application of a design pattern [76]. Past research conducted by us involved analyzing EMF models to detect Java EE anti patterns [62]. We also later developed SIMAID to detect Simulink anti patterns [65, 68] by looking for the intersection of clones of known anti patterns and target systems. All of these research ideas and tools are focused on detecting known/existing design patterns and/or anti patterns. The problem we focus on in this chapter is the detection of emergent patterns.

1.2.2 Model Clone Detection

Software clone detection involves finding identical or similar sets of software artifacts, which are termed "clones". Similarity is established by a variety of measures for different tools. Tools often have a similarity threshold criteria that must be met in order for two or more artifacts to be considered similar

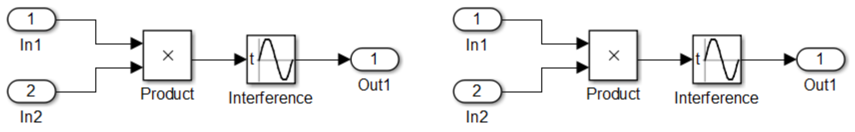


FIGURE 1.1 Example of an Exact/Identical, Type 1, Simulink Model Clone

enough to be identified as a clone. Software clones can exist due to a variety of reasons include a rush to deliver, bad reuse practices, unfamiliarity with cloning, and other reasons [39]. Software clones are not necessarily an indicator of poor software quality [37], and are important to identify regardless [35]. The majority of research and tooling involving software clone detection has been focused explicitly on source code clones [51, 53]. Only within the last decade, has model clone detection research begun to materialize [19]. Model clone detection is a form of model comparison [67] that discovers clones in software modeling artifacts. As we will be discussing model clone detection extensively in this chapter, it is necessary to define model clone types and overview existing model clone detection approaches.

1.2.2.1 Model Clone Types

Just as code clones can be categorized into different types based on their nature [53], so can model clones. It is generally accepted that there are four types of model clones, although the specific categorizations vary [4, 69, 70]. We now overview these as they are important concepts in understanding model clone detection's role and potential for model pattern mining. We describe them both textually and with contrived examples using Matlab Simulink[1] models.

1.2.2.1.1 Type 1 - Identical/Exact Model Clones

Identical, or "exact", type 1 model clones are those that are structurally identical to one another. They are identical except for aesthetic aspects such as layout / positioning, colour, formatting, et cetera. That is, it ignores those aspects in its comparison.

We present an example of type 1 identical Simulink model clone in Figure 1.1. Here we see two models that are completely identically, including labels, relationships, and block types. These models both take in two inputs, multiple them, apply a Sine Wave, and output the result.

1.2.2.1.2 Type 2 - Renamed Model Clones

Renamed, type 2 model, clones are those that are identical to each other as defined for type 1 identical model clones, except that type 2 model clones allow for differences in names and / or labels, attributes and / or values, and types and

1. https://www.mathworks.com/products/simulink.html

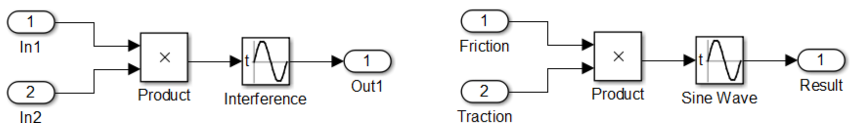


FIGURE 1.2 Example of a Renamed, Type 2, Simulink Model Clone

/ or parts [4, 70].

Figure 1.2 presents a contrived example of a type 2 renamed model clone. These models are identical to one another, except for that we see four of the five blocks have been renamed by the modeller. Specifically, the two inputs are renamed, the Interference block is now named Sine Wave, and the output block is now called Result.

1.2.2.1.3 Type 3 - Near-Miss Model Clones

A near miss, type 3, model clone is a model fragment that is one that is similar to another model fragment allowing for the same differences as type 2 model clones and additional structural modifications, such as adding or removing blocks / parts. Detectors typically use some form of threshold to indicate how "similar" two model fragments must be in order to be considered a clone. For example, allowing a percentage difference of 20%, that is, model clones that are 80% similar to one another.

Figure 1.3 demonstrates a sample type 3, near-miss, model clone. Here we illustrate two models that are similar to one another. In this case, however, the modeller made a slight modification compared to the original model at the top of the figure by adding an additional, Reciprocal Sqrt, block. Thus, a new block and additional line is present in the bottom model, causing them to be similar but not identical, near-miss clones. Their similarity percentage would be less than 100% but likely above a model clone similarity threshold.

1.2.2.1.4 Type 4 - Semantically Equivalent Clones

A type 4, semantically equivalent clone, is one that is structurally different enough that it may not be similar enough to be identified as a type 3 model clone but is semantically and behaviorally equivalent [1]. There are variety of reasons these may exist in software systems, including refactoring, coincidence, language constraints, and others [1, 70] .

We present a semantically equivalent, type 4, model clone in Figure 1.4. In this case, we have two models that are semantically equivalent, but potentially different enough from one another that a model clone detector may not find them to be within its structural similarity threshold. Specifically, we see a contrast in number of blocks and types of blocks. Both of these models add the input from In1 to itself, and take the square root of that number. The upper model does so using an addition block and (naively) takes the reciprocal of the reciprocal

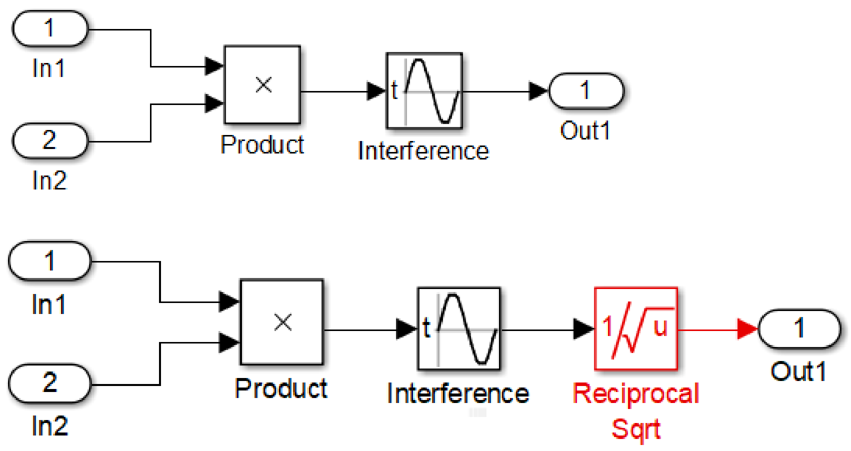


FIGURE 1.3 Example of a Near-Miss, Type 3, Simulink Model Clone

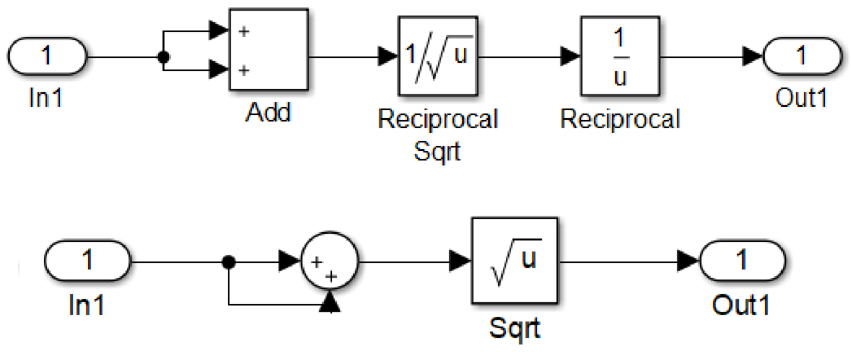


FIGURE 1.4 Example of a Semantically Equivalent, Type 4, Simulink Model Clone

square root. The bottom model uses Simulink's sum block to add the input from In1 to itself and takes the square root of that number directly.

1.2.2.2 Model Clone Detectors

Model clone detectors exist that employ varying means of detection and have the ability to detect model clones in a variety of modeling languages. Simulink model clone detector research is the most mature of all modeling languages [3, 18, 47, 48]. However, techniques are emerging for other modeling languages, including Stateflow models [14, 40] and UML models [6, 70, 71]. Techniques are also being devised by researchers to detect clones in metamodels, such as EMF models [7], and for model transformation languages [72].

We now discuss notable model clone detection approaches, which we categorize based on how they view, and/or the form they consider for comparing, modeling elements.

1.2.2.2.1 Graph-Matching Model Clone Detection

One of the first model clone detectors was ConQAT [19]. It detects Simulink models by considering their underlying graphical relationships. ConQAT calculates a graph label that encapsulates the modeling elements' information deemed important by them for similarity detection, which is block type specific. They employ a breadth-first search approach on the graph to find clone pairs, and subsequently cluster their clones. Similarly, Peterson developed the "Naive Clone Detector", which detects type 1, identical, Simulink model clones employing graph-based techniques and Simulink domain knowledge [47]. However, Peterson's approach uses a top-down approach instead of a breadth-first, and is implemented as a proprietary detector.

Pham et al. developed the eScan and aScan model clone detectors for detecting exact and approximate model clones, respectively [49]. Their identical model clones are detected by eScan through a consideration of size and graph node labels representing topology. To facilitate approximate, type 3, model clones, they allow for differences in those labels. Both of these tools are no longer available nor supported.

1.2.2.2.2 Text-based Model Clone Detection

The MQlone tool, developed by Störrle, detects UML model clones. MQlone works with a slightly different classification of model clone types, using the class types A, B, C, and D, and is capable of representing renamed clones. In their approach, they convert the XMI underlying representation of UML CASE models into Prolog representations[2]. In doing so, they are able to detect model clones using similarity metrics and static identifiers. Examples of metrics that MQlone considers are name distance, containment relationships of models, and size.

The Simone Simulink model clone detector detects Simulink model clones of types 1, 2, and 3 [3]. To do so, they focus on the underlying textual representations of Simulink models, and adapt the tried and tested source code clone detector, Nicad [54]. It is adapted in such a way that it is model sensitive and considers Simulink concepts including systems, lines, blocks, and whole models. Based on industrial feedback and systems being the main unit of organization within Simulink, Simone detects system level model clones. It has since been extended to work with Stateflow models [14], and UML behavioral models [6].

2. www.swi-prolog.org

1.3 MCPM - A CONCEPTUAL FRAMEWORK FOR USING MODEL CLONE DETECTION FOR PATTERN MINING

In this section, we discuss model clone detection's potential and possible role in the model analytics task of emergent pattern mining. Based on their work with industrial partners, Cordy et al. described their exploratory research extracting emergent patterns in Simulink models [2, 16]. They split up their project into three phases [16]: discovery, formalization, and application. They do not go into great detail about the phases themselves, but present their work in realizing them for their specific project. In this section, we formally refine / transform these phases to use accepted data mining terminology [23], which we use as the foundation for the MCPM conceptual framework with the intention that others can employ model clone detection for emergent pattern mining. To do this, we generalize these phases as steps within the framework, define the requirements for each of these steps, and discuss how existing research and tooling satisfies these requirements including some examples using Simulink models and our past experiences developing and employing a Simulink model clone detector, Simone [4]. That is, in each of the respective "Existing Research and Tooling" subsections for each step we are summarizing how existing work either meets or does not meet the respective requirements of the respective step.

We present the overall process within our conceptual MCPM framework in Figure 1.5. In contrast to Cordy's work [16], we use terms that are more consistent with that in the data mining and analytics literature [23]. The first phase in the process is Knowledge Discovery, which we break up into 4 sub steps: selection, preprocessing, transformation, and data mining through clustering. This involves employing model clone detection to identify models that are similar and cluster them together. The next phase in MCPM is Interpretation, which corresponds to Cordy's "formalization" phase. It also contains four sub steps. Interpretation can sometimes be considered by analysts as part of the knowledge discovery process. However, in MCPM we consider it more of a post processing and visualization step whereby clusters are analyzed by variation detection tools and techniques to find variation points, and then visualized to analysts. The next phase, Validation, involves validating the results of both the knowledge discovery and the interpretation / variation identification. The Application phase is the final phase, which focuses on applying the results of the model pattern mining in a form that is useful for analysts and end users.

1.3.1 Knowledge Discovery

In the MCPM framework, the knowledge discovery phase involves performing model clone detection on the models undergoing analysis. Model clone pairs are not useful in this context by themselves. The end goal of this phase is to yield not only models clone pairs, but model clone clusters, or model clone classes.

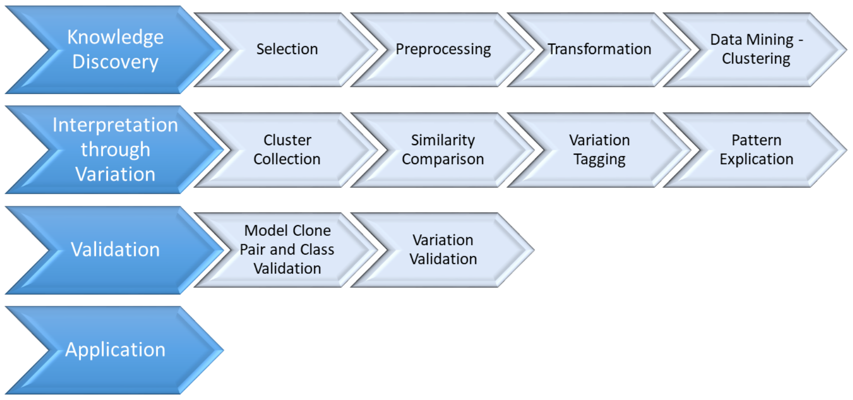


FIGURE 1.5 Overview of the Conceptual MCPM Framework

1.3.1.1 Knowledge Discovery - Requirements

The first requirement for this phase addresses the types of model clones that a detector is able to identify. Detection of type 1 and type 2 model clones may be useful for identifying rudimentary patterns from model instances that are identical or renamed. However, truly emergent and not-readily apparent model patterns would be discovered through detection of type 3, near-miss, model clones only, as verified through our experiences working with industrial partners [3, 16]. Type 4, semantically equivalent, model clones may also prove interesting in discovering patterns, but would likely be more applicable in refactoring scenarios, and would be challenging to parameterize as required in the next phase of our framework. Thus, a model clone detector would need to be able to detect near miss, type 3, model clones.

Since model clone pairs themselves are not useful in establishing an emergent pattern, a higher level categorization must take place in the mining phase. Specifically, a categorization of model clone clusters or classes that identify an abstraction of similar model clones beyond pairs. Not all code [53] nor model clone detectors [64] are able to do this. However, model clone clustering and/or classification is a requirement for a model clone detector to be used within the MCPM framework.

Scalability is always a concern when it comes to data mining [28]. From a model clone detection perspective, the more models that can be analyzed and mined, the more informative and "correct" the results will be. With the increasing size and complexity of software models being a forefront concern of model analytics and management, especially for large scale systems, this too is a key non-functional requirement when evaluating a model clone detectors' appropriateness for this task.

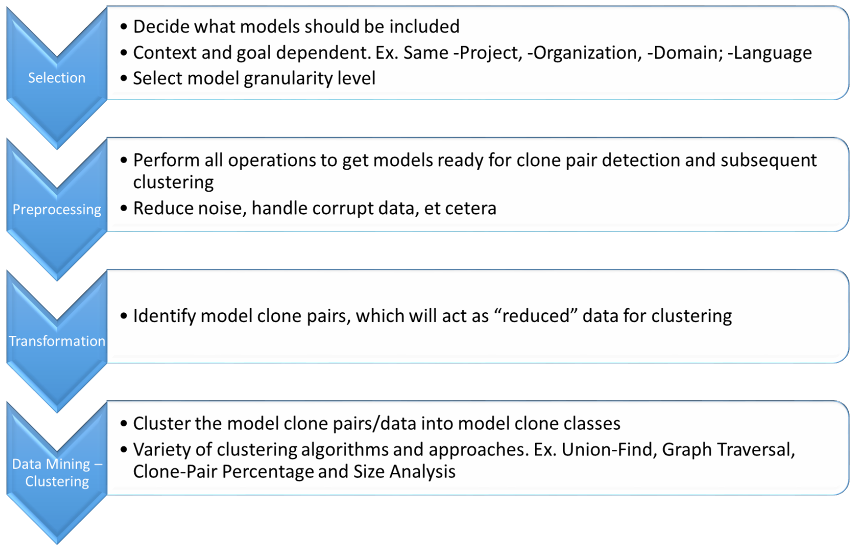

Selection
• Decide what models should be included
• Context and goal dependent. Ex. Same -Project, -Organization, -Domain; -Language
• Select model granularity level
Preprocessing
• Perform all operations to get models ready for clone pair detection and subsequent clustering
• Reduce noise, handle corrupt data, et cetera
Transformation
• Identify model clone pairs, which will act as "reduced" data for clustering
Data Mining – Clustering
• Cluster the model clone pairs/data into model clone classes
• Variety of clustering algorithms and approaches. Ex. Union-Find, Graph Traversal, Clone-Pair Percentage and Size Analysis

FIGURE 1.6 Summary of MCPM Knowledge Discovery Phase

1.3.1.2 Knowledge Discovery - Sub Steps

We break down this phase into four sub steps based on data mining practices [23]. We outline and summarize the sub steps in Figure 1.6, which we describe herein.

1.3.1.2.1 Selection

The selection step involves choosing what data will be analyzed [23]. For MCPM, this requires analysts to decide what models should be selected for emergent pattern mining. Much of this is context dependent. For example, if a specific organization is trying to codify and explicate emergent patterns based solely on their organization's past development for internal future reference, then they can select their own models. They may additionally, however, choose to include also models from the same domain, for example, an automotive company may choose to use their own models and all other automotive models for which they have access, such as open source models or models/software they have acquired through acquisition agreements. For more general analysis, for example by an academic or consultant, open source models from Github and Sourceforge, and model-specific repositories such as the Lindholmen Dataset [32] or MDEForge [10] can be included in the set of data. Additionally, this selection step is the time for analysts to decide on any subsets (models to include, in the case of MCPM), or variables within the data [23]. From a model perspective, this is dependent on the modeling language under consideration. For Simulink, for example, this can manifest itself into a question of granularity. Do analysts want to discovery emergent patterns at the model level, system level, or other level? For UML class models, this may be a question of package level patterns, relationship level patterns, or interface level patters.

1.3.1.2.2 Preprocessing

The preprocessing step in MCPM is for any models, or their data, that need to be manipulated and changed before analysis can occur. Each model clone detector must have the proper preprocessing in place so that, not only can model clone pairs be discovered, but also clustered. This can include noise reduction, deciding on what information is necessary to consider or ignore for mining, how to handle corrupt/missing data, and other aspects. For example, ConQAT flattens all Simulink models to remove their hierarchy and transforms Simulink blocks into labels that contain enough information necessary for clone detection [19]. The labels are dependent on the Simulink type of block being evaluated. Simone is another example of a Simulink model clone detector that preprocess its models and their data. In Simone's case, they preprocess the underlying Simulink model textual representations by normalizing, filtering, and sorting the text and its elements [3]. MQlone pre-processes its UML models of interest by transforming their XMI representation files into Prolog logic programs.

1.3.1.2.3 Transformation

Fayyad et al. define transformation as the process of reducing and projecting the data so that useful features can be found through the data mining process to follow [23]. This can also be thought of in terms of reducing variables under consideration or finding "invariant" data representations in our context. In the MCPM framework, transformation entails having a model clone detector use this step to identify model clone pairs, or some other immediate form, to facilitate model clone clustering / model clone class identification.

1.3.1.2.4 Data Mining - Clustering

This final step in the knowledge discovery phase involves clustering the model clones, or other reduced model clone data, into "similar" groups or classes. This can be accomplished through a variety of clustering algorithms and approaches, for example those discussed by Bishop [12]. In ConQAT's case, they discover model clone classes by devising a graph that represents clone pairs and having edges represent potential cloning relationships [19]. They then employ a combination of union-find structure analysis and graph traversal algorithms to cluster their clones. Simone generates model clone classes by clustering via the percentage differences among clone pairs and their sizes and uses connected component analysis. They then select the largest clone within the model clone class to use as a demonstration / representation of the class. MQlone does not perform clustering.

1.3.1.3 Knowledge Discovery - Existing Research and Tooling

When it comes to existing research and tooling, there are different model clone detectors to consider. ConQAT is currently unable to detect type 3, near-miss, model clones [63, 69]. So, although they do perform model clone clustering, they do not fulfill the MCPM requirements of detecting the necessary clone types. MQlone detects type 3, near-miss, model clones for UML models. While we did not find any indications that they currently cluster or create model clone classes [70, 71], there is no evidence suggesting MQlone cannot be extended to do so. With respect to MQlone's scalability, they have demonstrated its performance and ability to handle many clones, noting it is "mildly polynomial" with respect to model sizes [71]. Simone is another viable candidate for the MCPM framework. It is able to detect near-miss Simulink model clone pairs, and also cluster them in to model clone classes. Simone has been shown to have higher precision and recall than other Simulink model clone detectors [69]. From a scalability perspective, large model sets including industrial automotive sets and open-source systems with hundreds of Simulink systems were processed in minutes [3, 69].

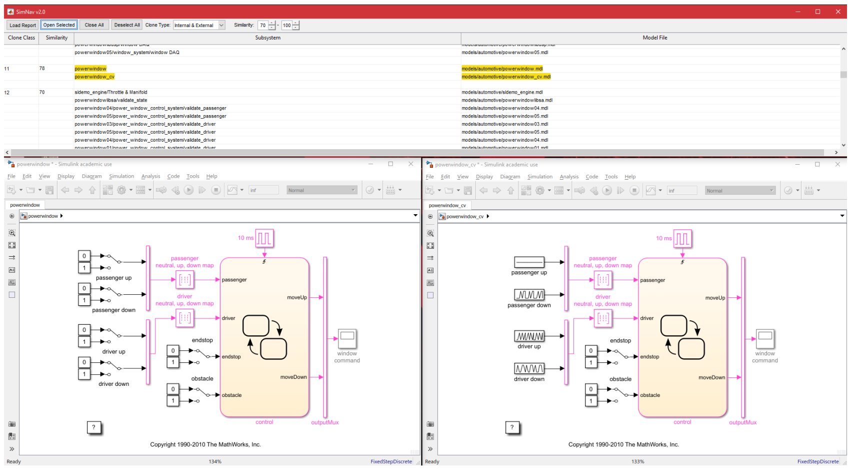
1.3.2 Interpretation through Variation

While these model clone classes can be thought of as direct representations of emergent model patterns, it still lacks applicability in that the patterns have not been explicated nor parameterized [16]. That is, we have examples of the pattern, but not its general form or how it can be used.

One illustration of this shortcoming is shown by the clone class navigation and visualization tool SimNav [50]. SimNav is a tool that is capable of interpreting the clone detection results of Simone and visualizing them directly in the Simulink environment for developers to view and interact with the identified classes. However, it still lacks a direct representation of variability. Figure 1.7 demonstrates SimNav's ability to select and display a model clone class to users, who are are still left with the specific task of determining if a pattern exists, and, if so, how it can be used or generalized.

1.3.2.1 Interpretation through Variation - Requirements

In order to effectively interpret model clone detection results in a meaningful manner for emergent pattern detection, any given approach must be able to apply some method to model variability explicitly. It is not sufficient to highlight that variability exists within a model cluster; an approach must provide representation of variability. This requirement can be accomplished in two main ways: 1) the construction of a common set of elements to represent the base pattern (elements not included in this base set should be marked as variants), or 2) the explicit marking of variation points in each model instance. Essentially, both approaches involve the tagging of the model, either in the model itself or some newly created


SimNav v2.0
Load Report
Open Selected
Close All
Deselect All
Clone Type: Internal & External
Similarity: 70 - 100
powerwindow * - Simulink academic use
powerwindow_cv * - Simulink academic use
File Edit View Display Diagram Simulation Analysis Code Tools Help
10 ms
passenger neutral, up, down map
driver neutral, up, down map
passenger up
passenger down
driver up
driver down
endstop
obstacle
passenger
driver
moveUp
moveDown
control
outputMux
window command
Copyright 1990-2010 The MathWorks, Inc.
Ready
134%
133%
FixedStepDiscrete

Clone Class	Similarity	Subsystem	Model File
		powerwindow05/window_system/window DAQ	models/automotive/powerwindow05.mdl
11	78	powerwindow	models/automotive/powerwindow.mdl
		powerwindow_cv	models/automotive/powerwindow_cv.mdl
12	70	sldemo_engine/Throttle & Manifold	models/automotive/sldemo_engine.mdl
		powerwindowlibsa/validate_state	models/automotive/powerwindowlibsa.mdl
		powerwindow04/power_window_control_system/validate_passenger	models/automotive/powerwindow04.mdl
		powerwindow05/power_window_control_system/validate_passenger	models/automotive/powerwindow05.mdl
		powerwindow03/power_window_control_system/validate_driver	models/automotive/powerwindow03.mdl
		powerwindow05/power_window_control_system/validate_driver	models/automotive/powerwindow05.mdl
		powerwindow04/power_window_control_system/validate_driver	models/automotive/powerwindow04.mdl

FIGURE 1.7 SimNav showing a detected clone class with two instances

variation model, to identify common and/or variable model elements.

The second requirement of interpreting model clone classes through variation is the ability to apply the variability patterns in a useful manner. One of the main applications of emergent model patterns is the ability to consolidate large numbers of similar, but not necessarily exact, models into one model file to address model maintenance issues. By representing all possible variants of a pattern in one single model file, model maintenance teams are required to maintain one model only now as opposed to many different instances of an observed pattern, which may be spread over many different files at different levels of hierarchies within the models.

The third requirement relates to scalability. Any appropriate approach suitable for the MCPM framework should generalize to both large models and large sets of models. A pattern between a given clone pair does not sufficiently improve interpretation. In contrast, a large set of instances in a model clone class, summarized by a single pattern, can drastically improve the maintenance and understanding of a large-scale system through model instance replacements.

In summary, the following are the requirements we have conceived of for the Interpretation through Validation phase of the MCPM framework. We propose that any candidate must demonstrate the ability to

1. model variability explicitly
2. apply variability patterns in a useful manner
3. scale to large model sets

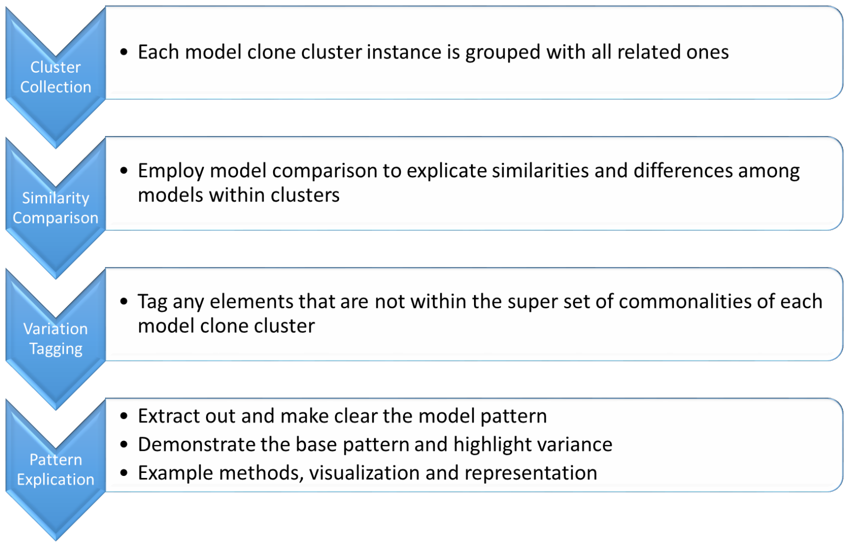

Cluster Collection
• Each model clone cluster instance is grouped with all related ones
Similarity Comparison
• Employ model comparison to explicate similarities and differences among models within clusters
Variation Tagging
• Tag any elements that are not within the super set of commonalities of each model clone cluster
Pattern Explication
• Extract out and make clear the model pattern
• Demonstrate the base pattern and highlight variance
• Example methods, visualization and representation

FIGURE 1.8 The Four Sub Steps to the Interpretation through Variation Phase of MCPM

1.3.2.2 Interpretation through Variation - Sub Steps

Based on these requirements, we further break down this phase of MCPM into 4 distinct steps as seen in Figure 1.8, which we further describe in this section. We use a simple example using Simulink models to demonstrate these steps.

1.3.2.2.1 Cluster Collection

The first step is to collect the model clone clusters in such a manner that each model instance is grouped along with all related instances, and all model instances are available for further analysis and manipulation. Since this phase may involve modifications to the models themselves, it may be prudent and/or necessary to work on copies of the original models to avoid unwanted overwriting of original artifacts. As we mentioned in requirement number 3 of this phase, the automatic clustering technique and tooling must be scalable enough for large model sets.

The running example we employ to illustrate this phase consists of four Simulink models. Each model takes in two numbers as input, performs some calculation (the variance), and produces an output. The pattern and variance are fairly obvious in the example but the process and techniques apply generally to any complexity of models. We introduce the four models in Figure 1.9.

1.3.2.2.2 Similarity Comparison

Within each collected clone cluster, the first goal is to determine the elements common to all instances. These form the basis of the pattern. At a high

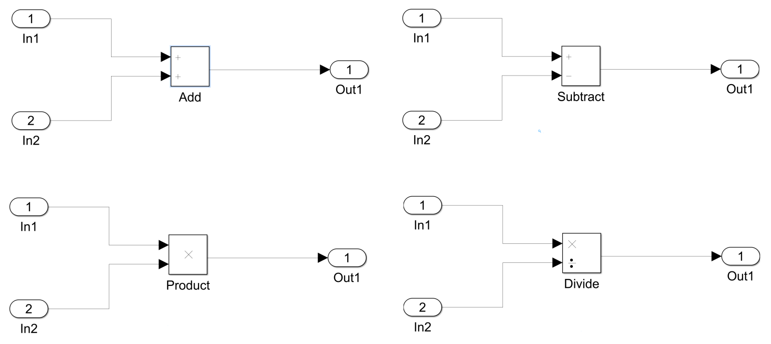


FIGURE 1.9 Running Example: Four Model Instances in the Same Model Clone Cluster

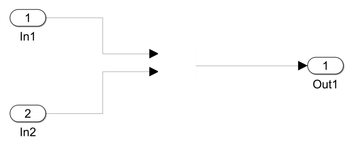


FIGURE 1.10 Running Example: The Superset of Model Elements in the Cluster of Instances

level, this is a simple application of model comparison tools to determine these commonalities. Model comparison involves identifying the similarities and differences among a set of models [38]. There are many approaches and tools for automatic model comparison to employ in this step [64, 67]. However, an approach or tool must be selected that is able to identify all the similarities explicitly, rather than differences only, and must be scalable, as we mentioned in our third requirement of this phase.

Independent of the chosen model comparison technique, the result of this comparison is a superset of model elements that are present in all model instances in the cluster. Regardless of the eventual application of each emergent pattern, this information may be of use to developers and analysts. The MCPM framework allows flexibility in what it considers a match, as it can accept a threshold of similarity between blocks. The default case would be 100% similarity, but there are potential applications where a lower similarity threshold may be desirable, such as renamed blocks or other near-miss scenarios. For example Schlie et al use 95% as their base threshold for similarity [57].

In our running example, the input and output ports, along with the associated connector lines, will be considered the set of similar elements to all instances, forming our superset, as we demonstrate in Figure 1.10.

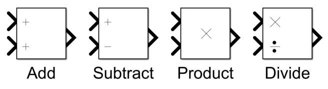


FIGURE 1.11 Running Example: The Variable Model Elements across the Cluster of Instances

1.3.2.2.3 Variation Tagging

In any particular instance model within a model clone cluster, any element that is not part of the previously identified superset of commonalities by the MCPM framework is a candidate for variation, which must be tagged through some method. This relates to requirement number 1 of being able to model variability explicitly. Any element that is not common to all instances represents one particular configuration of the pattern as observed by the respective model clone detector employed in the previous phase.

There are multiple forms we recommend for use within the MCPM framework. It can either be 1) based on the presence or absence of a model element, or 2) it can be based on a relative similarity. If the tagging is based on presence or absence, each different element would present a new variant of the pattern. If analysis reveals there is similarity (below the threshold chosen for inclusion in the superset of similar elements) between elements in two or more instances, these can be tagged by the framework implementation as alternatives rather than new variants. In both cases, variation tagging would, ideally, be a fully, but potentially semi, automatic process in order for it to scale to large model sets.

In our running example, the four mathematical operators, addition, subtraction, product, division, are the only elements not tagged as part of the similarity superset. Thus, they are the logical candidates for variation tagging. At a minimum each are tagged as a new variation of the model pattern due to being present. However, after additional analysis of the properties of the blocks, mainly number of inputs and outputs in this example, it can be reasonably concluded that each of them are similar enough to be considered variants of the same operation. Thus we identify four variants, as we exhibit in Figure 1.11.

1.3.2.2.4 Pattern Explication

The final sub step in this phase involves automatically explicating the variation/pattern in a meaningful way. This relates to the second requirement we identified for this phase. While this can take many forms in practice, each model pattern instance is an explication of the pattern, which as a minimum must demonstrate the existence of a base pattern and highlight the variance in some manner. The two main methods of explication within the MCPM framework are visualization and representation.

The first method is to create a visualization of the model pattern that identifies both the common elements and its variation points. In its simplest form, this is

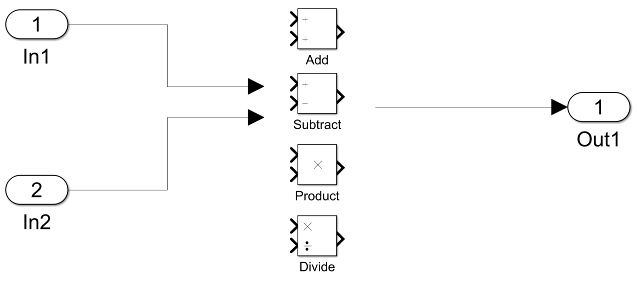


FIGURE 1.12 Running Example: A Sample Visualization of Variance

a merge of all model elements into a single model with duplicates of common elements removed. For our running example, a visualization of the pattern may appear as we show in Figure 1.12. This Simulink model contains all 4 variations of the pattern's instances. This abstraction is very explicit, and if implemented within the Simulink environment can allow modeller interaction with the pattern by selecting a specific instance.

The second method of explicating model patterns, which is more pragmatic, is to represent the pattern in some new format from which it can be validated and applied by analysts. In the next section we will discuss some existing implementations for doing this, but, at a high level, the goal is the creation of a single model file capable or representing the pattern and its variance. One such approach, which we apply to the running example, was introduced by us in our past work [5] in which we use Simulink Variant Subsystem Blocks[3]. Essentially all variants are contained in a special Simulink variant block that will connect the desired variant based on an environment variable selection only. For the running example, we present the resulting model in Figure 1.13. The Variant Subsystem block will contain four variants that can be selected by a modeller. Essentially, and in general, the model pattern is the whole model, including the commonalities, represented by all the static blocks, and the variants, represented by anything stored inside variant subsystem blocks.

1.3.2.3 Interpretation through Variation - Existing Research and Tooling

There exist several approaches that go beyond the identification of patterns shown in the Knowledge Discovery phase. These approaches make use of clustering and differencing techniques to explicitly represent variability in some form. It is from these approaches that we draw motivation for the Interpretation phase of MCPM.

Wille et al. apply variability mining to generic block-based modeling languages [79]. Specifically, they focus their work on Simulink and state charts.

3. https://www.mathworks.com/help/simulink/examples/variant-subsystems.html

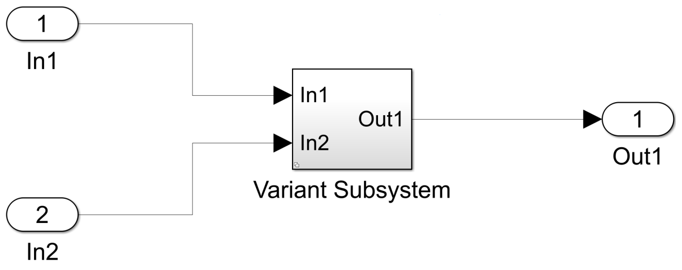


FIGURE 1.13 Running Example: A Sample Representation of Variance using Simulink Variant Subsystem Blocks

This approach is preliminary work that was later expanded upon by Schlie et al. in their technique to represent variability in Simulink models [57]. Their method proposes the use of a family model [33], wherein model elements are tagged as "mandatory" if they appear in all variants, "alternative" if they represent a variation point, and "optional" if they appear in one variant with no counterpart in the other. This is very similar to feature models in product line engineering [17]. Their Family Mining Framework consists of three distinct phases: compare, match, and merge. During the *compare* phase, models are compared using a comparison technique to generate a list of Compare Elements (CE), which are essentially a pair of similar elements with a normalized similarity value. During the *match* phase, a subset of all CEs is chosen by the algorithm as those that feature the *best matches*. Finally, during the *merge* phase, the models are merged into a single model based on a user specified mapping function which determines for each element if it is mandatory, alternative, or optional. Schlie et al. also present an expansion of this work where they apply the Family Mining Framework to Reverse Signal Propagation Analysis (RSPA) [56]. RSPA consists of three phases: signal set generation, comparing and clustering, and cluster optimization. During *signal set generation*, their approach traverses each hierarchical layer of the model to generate a set of outgoing signals present on that layer. These signal sets are then used as a basis for *comparing and clustering*, in which blocks associated with varying signals are clustered into sets of preliminary clusters. These preliminary clusters may contain some intersections, and thus *cluster optimization* is used to improve the clustering iteratively until no more intersections exist. Essentially, rather than relying on other similarity metrics to cluster blocks together, RSPA uses the unique signals and propagates them through the models at various levels to determine similarity and clustering, which is then used to determine variation points. Willie et al. [78] also devised an approach for configurable detection of variability relations for model variants at the block level. Their work would be helpful in an MCPM framework by providing guidelines for variability mining interpretation and preprocessing.

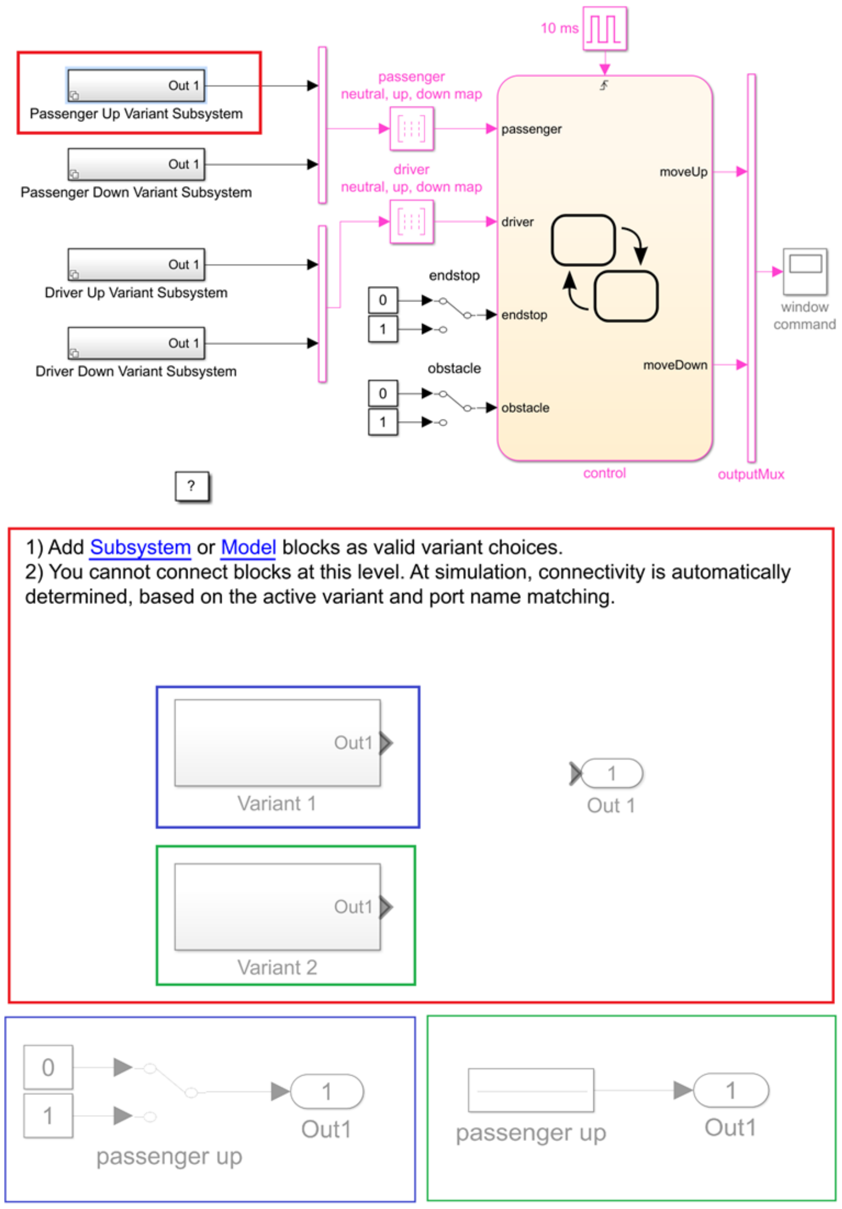
Our past work involved developing a method of using detected clusters of clones (clone classes) to model variation points in Simulink models [5]. Our approach asserts that a clone class with limited variability between each individual model is best represented by a single model using built-in Simulink Variant Subsystem Blocks for each variation point. Each original version of the model can then be recreated by a modeler by setting environment variables to configure each choice to match the original subsystem, while allowing maintenance to be performed on one model file instead of the number of models originally contained in the detected clone class. This approach accounts for five different types of variability: block, input/output, function, layout, and subsystem naming, with each being handled in a similar manner. This technique can be applied to the model clone class detected and shown in Figure 1.7 to create a single model file capable of representing both models. The original variance between the two instances can be seen on the left of the models; each has its own method of producing input signals. We present the resulting variability model in Figure 1.14, with each expanded subsystem expanded in detail. One of the main advantages of this technique is its ability to scale to large-scale systems, both in terms of model complexity and size of model sets. The other techniques we discussed were applied mainly to pairwise variants, whereas this technique can combine arbitrarily large sets of model variants into a single variability model.

1.3.3 Validation

This phase of the conceptual MCPM framework involves framework implementers validating the results of the previous two phases. Specifically, it is necessary to validate the results of the model clone detection and clustering, and the pattern interpretation of those clusters. Problems in mining can sometimes produce results that seem to be predictive and useful, but actually are misleading and cannot be reproduced [31]. This can occur at various phases within pattern mining, so we thus evaluate each phase.

1.3.3.1 Validation of the Model Clone Pair and Model Clone Clusters

As model clone detection tooling and research is still emerging, there are limited techniques for evaluating and validating model clone detection results. In our past work, we developed a framework, MuMonDE, for evaluating model clone detectors that employs automatic mutation analysis to ascertain correctness and recall [69]. It mutates the models using a predefined mutation taxonomy [66] to find out if the model clones that should be detected are detected. Currently it has been employed by researchers to evaluate Simulink model clone detectors only. Störrle has performed some model clone evaluation experiments of their own to evaluate MQlone by manually seeding clones within existing model bases and seeing if they are detected [71]. Regardless of how validation is performed by an MCPM framework implementer, the general idea is to ensure the model clone detector is exhibiting high precision and recall in their model clone detection executions.



1) Add Subsystem or Model blocks as valid variant choices.
2) You cannot connect blocks at this level. At simulation, connectivity is automatically determined, based on the active variant and port name matching.

FIGURE 1.14 Applying Alalfi et al's. technique [5] to represent the variants in the model clone class shown in Figure 1.7

TABLE 1.1 Running Example: Sample Test Case Values for 4 Math Function Models

Addition Test Cases			**Subtraction Test Cases**		
Input 1	**Input 2**	**Output**	**Input 1**	**Input 2**	**Output**
0	1	1	0	0	0
1	1	2	1	0	1
2	1	3	2	0	2
...			...		
Multiplication Test Cases			**Division Test Cases**		
Input 1	**Input 2**	**Output**	**Input 1**	**Input 2**	**Output**
2	1	2	1	1	1
2	2	4	2	1	2
2	3	6	3	1	3
...			...		

1.3.3.2 Validation of the Interpretation through Variation

With respect to validating results obtained by interpreting the observed variations within the MCPM framework, there are two main approaches that can be taken: a test-driven approach and a reverse engineering approach, both of which we describe in detail.

The test-driven approach applies mainly to any application of representation of variation, that is, any time a variant model is created regardless of technologies used. The general idea is that if a set of tests exist for the original corpus of models prior to model clone detection, clustering, and variant modeling, the newly created variant model should be capable of being configured in such a way so as to successfully pass each of the original test cases. This is analogous to the idea of finding semantically equivalent source code via testing [34]. Consider the running example from the previous section. Each model would have its own set of unique test cases to ensure it functions as required, and would have been able to independently pass all of the required tests. Table 1.1 provides a small set of example tests. After the creation of a variant model, for example the one we presented in Figure 1.13, to validate the correctness of the variant model, it must be able to be configured by a tester to be able to pass all four sets of tests. While this is not a guarantee of correctness of the newly created variant model, it provides a high level of confidence in its ability to represent the original behavior of all original pattern elements.

The second method of validating the results of interpretation through variation within the MCPM is through the use of reverse engineering techniques. The goal of this type of validation is to be able to accurately recreate the original models that were used to create the resulting pattern. For example, by means of model transformations to recreate the original models. This can result in a high level of confidence in the representation of the pattern. The added benefit of using reverse engineering to validate pattern representation is that it applies to both representations as a variant model, as well as a more simplified visualization of variance. The reverse engineering technique works by applying, in reverse order, the same steps we recommended to create the representation or visualization. If it is possible to return to the original models that form the pattern, it becomes evident the new model pattern representation/visualization has not lost any of its original semantics. However, if it is not possible to obtain the original models from the resulting visualization or representation, this means there might be some loss of information, thus reducing the overall confidence in the interpretation of the pattern.

While there may exist other application and domain specific methods of validating variation results, the two techniques we discussed here present methods that generalize to most cases. Further, these techniques are sufficiently scalable to apply to large scale systems without additional effort, provided the required inputs (test files or list of applied steps) exist previously.

1.3.3.3 Validation of Pattern Quality

A potential third area of validation is the evaluation of the quality of any detected patterns. The previous validations deal largely with correctness and the ability to express patterns. However, the "quality" of the patterns is still unknown to analysts. In its current state, we do not consider this aspect of validation within scope for the MCPM conceptual framework as the MCPM is currently concerned mainly with identifying emergent patterns and expressing them in a form conducive to evaluation and application. Quality evaluation and filtering, sorting, and/or ranking of patterns is interesting from a post processing perspective, but not part of the core MCPM.

1.3.4 Application

While the identification of valid emergent model patterns through model clone detection presents useful information, the MCPM framework posits that a particular application of the obtained knowledge is important to solidify its contributions. As such, this section presents our recommendations on potential applications of findings in industrial settings, including integration into work flows and tool chains.

The first potential application, and perhaps the simplest, is the incorporation of pattern visualization into the work flow of model development and model-driven engineering. Being aware of the existence of model patterns in a collection of models yields its own benefits in that developers are able to draw reasonable conclusions about their presence and adapt usual working conditions to make changes. While this seems abstract, it fits the goals of the MCPM framework and allows the domain experts to use the knowledge to improve their models as appropriate. Regarding pattern usefulness, the objective of MCPM is solely to identify emergent patterns. Thus, evaluation of the usefulness of the patterns that emerge is beyond the scope of the framework. Analysts can perform filtering, ranking, and sorting of the emergent patters depending on their intended use cases. One of the primary ways this information is applied is via the identification of potential faults. In some situations, an expert can be left asking the question ”Is this variance point intentional, or have we found a fault?”. By applying the MCPM framework to a project periodically, relevant information can be revealed to improve overall software quality.

The second application combats the model management problem discussed earlier, and that is one of the focuses of this book. Determining the existence of model patterns in model sets provides candidates for model merging and retaining variance through metrics, allowing analysts to maintain only one model rather than the large number of original models. This process takes numerous forms as we discussed, but generally presents the outcome of a reduced number of models that are still capable of expressing the original functionality of the full set of models. Within Simulink, for example, this can be done through the use of Subsystem Variant blocks [5] or by creating tagged models with aspects similar to software product lines [57]. Regardless of approach, the introduction of variant models decreases the total number of models in the set, allowing for improved maintainability.

1.4 SUMMARY OF CHALLENGES AND FUTURE DIRECTIONS

The main challenges faced by the MCPM framework are the full automation of variant model creation, effective visualization and explication of model patterns, post validation of emergent pattern quality, and the practitioner adoption issue. Each issue presents an opportunity for future work, which we discuss further.

1.4.1 Automatic Variant Model Creation

We previously developed an approach [5] that is only semi-automated as a proof of concept. However, there is still no current method of creating a full set of variability models that work directly within Simulink. While the approach presents meaningful results, the manual interaction creates opportunity for error, and does not scale well to large scale systems, where the approach is most needed.

Schlie et al. present an approach [57] that deals with tagging elements in a Family model as mandatory, alternative, or optional. This does not translate directly to a Simulink model that can be used explicitly. Rather, it is merely an additional representation to present variability. While this approach scales better than the previous one, the separation from the application domain by representing the models outside of Simulink poses a pragmatic problem.

Both of these approaches demonstrate promise in variant modeling, however the development of a fully automated approach capable of scaling to large scale systems is still necessary and an open challenge to advance the application of the MCPM framework.

1.4.2 Effective Visualization/Explication

In order to use the results of model clone detection to find model patterns, an effective way of explicating these model patterns is needed. When working with experts in graphical modeling, this likely will take the form of visualization in order to maintain the higher level abstractions afforded by modeling. However, there is no strong forerunner in the visualization of explicit model patterns. While we presented SimNav [50] as a method of visualizing the model clone classes, the approach focuses too specifically on clone classes, and not the underlying patterns.

In order to advance the application of the MCPM framework, further work in the effective visualization and explication of model patterns is required, specifically with some form of automated tool support.

1.4.3 Validating Pattern Quality

Currently, the MCPM framework does not address the quality of the observed patterns. Within the framework, detected patterns are always included in the resulting pattern set, as they are emergent. However this may not always be the best course of action. In order to improve the effectiveness of applying patterns, it may be necessary to include a post processing method for validating pattern quality as part of the MCPM framework in its future implementations.

1.4.4 Practitioner Adoption

The final challenge faced currently by the MCPM framework concept is the potential reluctance of practitioners to adopt elements of the MCPM framework. While this is a problem that is faced in many domains, and possibly related to the solutions to the previous two issues, it is still worth noting as an ongoing challenge in the field. Industrial developers of models need to be on board with the concept of applying model clone detection to observe and make use of model patterns for the framework to advance and gain traction. While we do not have any specific approaches to overcome this hurdle, it is a challenge to the community to work closely with practitioners to ensure their buy-in and adoption of research technologies and tooling. One aspect of this challenge would be some sort of ranking or feedback system associated with any discovered emergent patterns to ensure quality. That is, a way of having practitioners review and rate the patterns to help with "good" and "bad" emergent pattern identification.

1.5 CONCLUSION

In this chapter, we have investigated the possibility of employing model clone detection as a tool for extracting emergent model patterns from large model repositories. We defined a conceptual framework, MCPM, that can be realized and used by analysts and researchers to follow in order to use model clone detection for this purpose. We break down the details of each of MCPM's four phases, provide examples to follow, and describe how current research and tooling fits its requirements. We additionally outline current challenges to help set the path for researchers interested in this area. It is our intention for this chapter to help pave the way for more advanced model analytics allowing for the identification of emergent model patterns in large software systems, thus improving model driven engineering on the whole.

References

[1] Bakr Al-Batran, Bernhard Schätz, and Benjamin Hummel. Semantic clone detection for model-based development of embedded systems. In *International Conference on Model Driven Engineering Languages and Systems*, pages 258–272. Springer, 2011.

[2] Manar H Alalfi, James R Cordy, and Thomas R Dean. Analysis and clustering of model clones: An automotive industrial experience. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 375–378. IEEE, 2014.

[3] Manar H. Alalfi, James R. Cordy, Thomas R. Dean, Matthew Stephan, and Andrew Stevenson. Models are code too: Near-miss clone detection for Simulink models. In *ICSM*, pages 295–304, 2012.

[4] Manar H Alalfi, James R Cordy, Thomas R Dean, Matthew Stephan, and Andrew Stevenson. Near-miss model clone detection for simulink models. In *Proceedings of the 6th International Workshop on Software Clones*, pages 78–79. IEEE Press, 2012.

[5] Manar H Alalfi, Eric J Rapos, Andrew Stevenson, Matthew Stephan, Thomas R Dean, and James R Cordy. Semi-automatic identification and representation of subsystem variability in simulink models. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 486–490. IEEE, 2014.

[6] Elizabeth Antony, Manar H. Alalfi, and James R. Cordy. An Approach to Clone Detection in Behavioural Models. In *International Working Conference in Reverse Engineering*, pages 472–476, 2013.

[7] Ö Babur. Clone detection for ecore metamodels using n-grams. In *International Conference on Model-Driven Engineering and Software Development*, 2018.

[8] Zsolt Balanyi and Rudolf Ferenc. Mining design patterns from c++ source code. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 305–314. IEEE, 2003.

[9] Demis Ballis, Andrea Baruzzo, and Marco Comini. A rule-based method to match software patterns against uml models. *Electronic Notes in Theoretical Computer Science*, 219:51–66, 2008.

[10] Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Amleto Di Salle, Ludovico Iovino, and Alfonso Pierantonio. MDEForge: an Extensible Web-Based Modeling Platform. In *International Workshop on Model-Driven Engineering on and for the Cloud*, pages 66–75, 2014.

[11] Federico Bergenti and Agostino Poggi. Idea: A design assistant based on automatic design pattern detection. In *Proceedings of the 12th international conference on Software Engineering and Knowledge Engineering*, pages 336–343. Springer-Verlag, 2000.

[12] Christopher Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[13] William H Brown, Raphael C Malveau, Hays W McCormick, and Thomas J Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[14] Jian Chen, Thomas R Dean, and Manar H Alalfi. Clone detection in matlab stateflow models. *Software Quality Journal*, 24(4):917–946, 2016.

[15] James O Coplien. Software design patterns: common questions and answers. *The Patterns Handbook: Techniques, Strategies, and Applications*, pages 311–320, 1998.

[16] James R Cordy. Submodel pattern extraction for simulink models. In *International Software Product Line Conference*, pages 7–10, 2013.

[17] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In *International Conference on Software Product Lines*, pages 266–283. Springer, 2004.

[18] F. Deissenboeck, B. Hummel, E. Juergens, B. Schaetz, S. Wagner, J.-F. Girard, and S. Teuchart. Clone detection in automotive model-based development. In *ICSE*, pages 603–612, 2009.

[19] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Bernhard Schätz, Stefan Wagner, Jean-François Girard, and Stefan Teuchert. Clone detection in automotive model-based development. In *30th international conference on Software engineering*, pages 603–612. ACM, 2008.

[20] Patricia Derler, Edward A Lee, and Alberto Sangiovanni Vincentelli. Modeling cyber–physical systems. *Proceedings of the IEEE*, 100(1):13–28, 2012.

[21] Giuseppe Di Fatta, Stefan Leue, and Evghenia Stegantova. Discriminative pattern mining in software fault detection. In *Proceedings of the 3rd international workshop on Software quality assurance*, pages 62–69. ACM, 2006.

[22] Jing Dong, Yajing Zhao, and Tu Peng. A review of design pattern mining techniques. *International Journal of Software Engineering and Knowledge Engineering*, 19(06):823–855, 2009.

[23] Usama M Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, et al. Knowledge discovery and data mining: Towards a unifying framework. In *KDD*, volume 96, pages 82–88, 1996.

[24] Eduardo Fernandez-Buglioni. *Security patterns in practice: designing secure architectures using software patterns*. John Wiley & Sons, 2013.

[25] Rahma Fourati, Nadia Bouassida, and Hanêne Ben Abdallah. A metric-based approach for anti-pattern detection in uml designs. In *Computer and Information Science 2011*, pages 17–33. Springer, 2011.

[26] Michael R Garey. A guide to the theory of np-completeness. *Computers and intractability*, 1979.

[27] Manjari Gupta, Rajwant Singh Rao, Akshara Pande, and AK Tripathi. Design pattern mining using state space representation of graph matching. In *International Conference on Computer Science and Information Technology*, pages 318–328. Springer, 2011.

[28] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[29] Neil B Harrison, Paris Avgeriou, and Uwe Zdun. Using patterns to capture architectural decisions. *IEEE software*, 24(4), 2007.

[30] Ahmed E Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57. IEEE, 2008.

[31] Douglas M Hawkins. The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12, 2004.

[32] Regina Hebig, Truong Ho Quang, Michel RV Chaudron, Gregorio Robles, and Miguel Angel Fernandez. The quest for open source projects that use uml: mining github. In *International Conference on Model Driven Engineering Languages and Systems*, pages 173–183. ACM, 2016.

[33] Sönke Holthusen, David Wille, Christoph Legat, Simon Beddig, Ina Schaefer, and Birgit Vogel-Heuser. Family model mining for function block diagrams in automation software. In *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2*, pages 36–43, 2014.

[34] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments

via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 81–92. ACM, 2009.

[35] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *31st International Conference on Software Engineering*, pages 485–495. IEEE, 2009.

[36] Huzefa Kagdi, Jonathan I Maletic, and Bonita Sharif. Mining software repositories for traceability links. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pages 145–154. IEEE, 2007.

[37] Cory Kapser and Michael W Godfrey. " cloning considered harmful" considered harmful. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 19–28. Citeseer, 2006.

[38] D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Model comparison: a foundation for model composition and model transformation testing. In *Proceedings of the International Workshop on Global Integrated Model Management*, pages 13–20. ACM, 2006.

[39] R. Koschke. Survey of research on software clones. *Duplication, Redundancy, and Similarity in Software*, pages 1–24, 2006.

[40] Mythili Aravida Kumar. Efficient weight assignment method for detection of clones in state flow diagrams. *Journal of Software Engineering Research and Practices*, 4(2):12–16, 2014.

[41] Jürgen Lind. Patterns in agent-oriented software engineering. In *International Workshop on Agent-Oriented Software Engineering*, pages 47–58. Springer, 2002.

[42] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *ACM SIGSOFT Software Engineering Notes*, 30(5):296–305, 2005.

[43] Gaurab Luitel, Matthew Stephan, and Daniela Inclezan. Model Level Design Pattern Instance Detection Using Answer Set Programming. In *International Workshop on Modeling in Software Engineering (MISE)*, MiSE '16, pages 13–19, New York, NY, USA, 2016. ACM.

[44] Robert C Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[45] Jukka Paakki, Anssi Karhinen, Juha Gustafsson, Lilli Nenonen, and A Inkeri Verkamo. Software metrics by architectural pattern mining. In *Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress)*, pages 325–332. Kluwer Beijing, China, 2000.

[46] Akshara Pande, Manjari Gupta, and AK Tripathi. Design pattern mining for gis application using graph matching techniques. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 3, pages 477–482. IEEE, 2010.

[47] Hauke Petersen. Clone detection in Matlab Simulink models. Master's thesis, Technical University of Denmark, 2012, iMM-M. Sc.-2012-02, 2012.

[48] Nam H Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M Al-Kofahi, and Tien N Nguyen. Complete and accurate clone detection in graph-based models. In *ICSE*, pages 276–286, 2009.

[49] N.H. Pham, H.A. Nguyen, T.T. Nguyen, J.M. Al-Kofahi, and T.N. Nguyen. Complete and accurate clone detection in graph-based models. In *International Conference on Software Engineering (ICSE)*, pages 276–286, 2009.

[50] E. J. Rapos, A. Stevenson, M. H. Alalfi, and J. R. Cordy. Simnav: Simulink navigation of model clone classes. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 241–246, 2015.

[51] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[52] Dirk Riehle and Heinz Züllighoven. Understanding and using patterns in software development.

Theory and practice of object systems, 2(1):3–13, 1996.

[53] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *QueenâĂŹs School of Computing TR*, 541(115):64–68, 2007.

[54] C.K. Roy and J.R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *ICPC*, pages 172–181, 2008.

[55] H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *14th IEEE International Conference on Program Comprehension(ICPC)*, volume 00, pages 84–88, 06 2006.

[56] Alexander Schlie, David Wille, Loek Cleophas, and Ina Schaefer. Clustering variation points in matlab/simulink models using reverse signal propagation analysis. In *International Conference on Software Reuse*, pages 77–94. Springer, 2017.

[57] Alexander Schlie, David Wille, Sandro Schulze, Loek Cleophas, and Ina Schaefer. Detecting variability in matlab/simulink models: an industry-inspired technique and its evaluation. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume A*, pages 215–224. ACM, 2017.

[58] Douglas C Schmidt, Mohamed Fayad, and Ralph E Johnson. Software patterns. *Communications of the ACM*, 39(10):37–39, 1996.

[59] Abhishek B Sharma, Franjo Ivančić, Alexandru Niculescu-Mizil, Haifeng Chen, and Guofei Jiang. Modeling and analytics for cyber-physical systems in the age of big data. *ACM SIGMETRICS Performance Evaluation Review*, 41(4):74–77, 2014.

[60] Nija Shi and Ronald A Olsson. Reverse engineering of design patterns from java source code. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 123–134. IEEE, 2006.

[61] Michael Stal. Using architectural patterns and blueprints for service-oriented architecture. *IEEE software*, 23(2):54–61, 2006.

[62] M. Stephan. Detection of Java EE EJB antipattern instances using framework-specific models. Master's thesis, University of Waterloo, 2009.

[63] M. Stephan, M.H. Alafi, A. Stevenson, and J.R. Cordy. Towards qualitative comparison of simulink model clone detection approaches. In *International Workshop on Software Clones (IWSC)*, pages 84–85, 2012.

[64] M. Stephan and J. R. Cordy. A survey of methods and applications of model comparison. Technical Report 2011-582 Rev. 3, Queen's University, 2012.

[65] M. Stephan and J. R. Cordy. Identification of Simulink model antipattern instances using model clone detection. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 276–285, Sept 2015.

[66] Matthew Stephan, Manar Alalfi, and James R. Cordy. Towards a taxonomy for simulink model mutations. In *International Conference on Software Testing, Verification, and Validation 2014 (ICST) – Mutation Workshop*, pages 206–215, 2014.

[67] Matthew Stephan and James R Cordy. A survey of model comparison approaches and applications. In *Modelsward*, pages 265–277, 2013.

[68] Matthew Stephan and James R. Cordy. Identifying instances of model design patterns and antipatterns using model clone detection. In *International Workshop on Modelling in Software Engineering*, pages 48–53, 2015.

[69] Matthew Stephan and James R Cordy. Mumonde: A framework for evaluating model clone detectors using model mutation analysis. *Software Testing, Verification and Reliability*, page e1669, 2018.

[70] Harald Störrle. Towards clone detection in uml domain models. *Software & Systems Modeling*, 12(2):307–329, 2013.

[71] Harald Störrle. Effective and efficient model clone detection. In *Software, Services, and Systems*, pages 440–457. Springer, 2015.

[72] Daniel Strüber, Vlad Acreţoaie, and Jennifer Plöger. Model clone detection for rule-based model transformation languages. *Software & Systems Modeling*, pages 1–22, 2017.

[73] Paolo Tonella and Mariano Ceccato. Aspect mining through the formal concept analysis of execution traces. In *11th Working Conference on Reverse Engineering*, pages 112–121. IEEE, 2004.

[74] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T Halkidis. Design pattern detection using similarity scoring. *IEEE transactions on software engineering*, 32(11), 2006.

[75] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.

[76] Sven Wenzel. Automatic detection of incomplete instances of structural patterns in uml class diagrams. *Nordic Journal of Computing*, 12(4):379, 2005.

[77] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85, 2014.

[78] David Wille, Önder Babur, Loek Cleophas, Christoph Seidl, Mark van den Brand, and Ina Schaefer. Improving custom-tailored variability mining using outlier and cluster detection. *Science of Computer Programming*, 163:62–84, 2018.

[79] David Wille, Sandro Schulze, Christoph Seidl, and Ina Schaefer. Custom-tailored variability mining for block-based languages. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 271–282. IEEE, 2016.

[80] Tao Xie and Jian Pei. Mapo: Mining api usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 54–57. ACM, 2006.

[81] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie Van Deursen. Mining software repositories to study co-evolution of production & test code. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 220–229. IEEE, 2008.

[82] Liming Zhu, Muhammad Ali Babar, and Ross Jeffery. Mining patterns to support software architecture evaluation. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, pages 25–34. IEEE, 2004.